DOI:
10.1016/j.tcs.2015.06.050

Document Version
Peer reviewed version

Order-Preserving Indexing[☆]

Maxime Crochemore[a], Costas S. Iliopoulos[a], Tomasz Kociumaka[b], Marcin Kubica[b], Alessio Langiu[c], Solon P. Pissis[a], Jakub Radoszewski[b], Wojciech Rytter[b], Tomasz Waleń[b]

[a] *Department of Informatics, King's College London, Strand, London, WC2R 2LS, United Kingdom*
[b] *Faculty of Mathematics, Informatics and Mechanics, University of Warsaw, Stefana Banacha 2, 02-097 Warsaw, Poland*
[c] *Institute for Coastal Marine Environment of the National Research Council (IAMC-CNR), Unit of Capo Granitola, Via del Faro no. 3, 91021 Granitola, TP, Italy*

Abstract

Kubica et al. (*Information Processing Letters*, 2013) and Kim et al. (*Theoretical Computer Science*, 2014) introduced order-preserving pattern matching: for a given text the goal is to find its factors having the same 'shape' as a given pattern. Known results include a linear-time algorithm for this problem (in case of polynomially-bounded alphabet) and a generalization to multiple patterns. We propose an index that enables order-preserving pattern matching queries in time proportional to pattern length. The index can be constructed in $O(n \log\log n)$ expected time or in $O(n \log^2 \log n / \log\log\log n)$ worst-case time. It is an incomplete order-preserving suffix tree which may miss a single edge label at each branching node. For most applications such incomplete suffix trees provide the same functional power as the complete ones. We show a number of their applications, including computation of longest common factors, longest previously occurring factors and squares in a string in the order-preserving setting. We also give an $O(n\sqrt{\log n})$-time algorithm constructing complete order-preserving suffix trees.

Keywords: order-preserving matching, order-preserving indexing, suffix tree

1. Introduction

We consider pattern matching and repetition discovery problems in the order-preserving setting. In the *order-preserving pattern matching problem* we look for consecutive fragments of a text which have the same relative order of letters as a pattern. This problem was introduced independently by Kim et al. [1] and Kubica et al. [2]. Applications of the order-preserving setting include detecting trends in time series, which appear naturally e.g. when considering the stock market or melody matching of two musical scores; see [1].

The study of order-preserving model evolved from the combinatorial study of patterns in permutations. The latter is focused on pattern avoidance, that is, counting the number of permutations not containing a subsequence order-isomorphic to a given pattern. Note that here the subsequences need not to be consecutive. The first results were given by Knuth [3] (avoidance of 312), Lovász [4] (avoidance of 213) and Rotem [5] (avoidance of both 231 and 312). Currently this is a very active field of research; a dedicated International Permutation Patterns Conference has been held annually since 2003. On the algorithmic side, pattern matching in permutations (as a subsequence) was shown to be NP-complete [6]. A number of polynomial-time algorithms for special cases of patterns were developed [7, 8, 9] and very recently an FPT algorithm parameterized by the length of the pattern was proposed by

[☆] A preliminary version of this paper appeared in the Proceedings of the String Processing and Information Retrieval Symposium 2013 (SPIRE 2013), pp. 84–95, 2013.

Email addresses: maxime.crochemore@kcl.ac.uk (Maxime Crochemore), c.iliopoulos@kcl.ac.uk (Costas S. Iliopoulos), kociumaka@mimuw.edu.pl (Tomasz Kociumaka), kubica@mimuw.edu.pl (Marcin Kubica), alessio.langiu@iamc.cnr.it (Alessio Langiu), solon.pissis@kcl.ac.uk (Solon P. Pissis), jrad@mimuw.edu.pl (Jakub Radoszewski), rytter@mimuw.edu.pl (Wojciech Rytter), walen@mimuw.edu.pl (Tomasz Waleń)

Guillemot and Marx [10]. A survey by Bruner and Lackner [11] lists further algorithmic results related to permutation patterns.

The (consecutive) order-preserving model was first studied by Kim et al. [1] and Kubica et al. [2]. In each of these papers an $O(n + m \log m)$-time algorithm for pattern matching in this model is presented, where n is the length of the text and m is the length of the pattern. Under a natural assumption that the characters of the pattern can be sorted in linear time, the algorithms can be implemented in $O(n + m)$ time. Several alternative solutions for order-preserving pattern matching problem, including practical implementations, have been proposed recently; see [12, 13, 14, 15, 16]. An algorithm for order-preserving matching with mismatches was published by Gawrychowski and Uznański [17]. A multiple-pattern matching algorithm based on the algorithm of Aho and Corasick was developed by Kim et al. [1]. Other studied problems in the order-preserving model include prefix tables [18, 16], periods, borders, and covers [16]. Order-preserving matching in the context of ternary order relations was recently studied in [19]. Also some combinatorial results concerning order-preserving squares have been obtained [20].

We introduce the problem of indexing for order-preserving pattern matching, in which one needs to preprocess a text to enable fast order-preserving pattern matching queries. In the literature there are a number of results for indexing in a related model of *parameterized pattern matching*. This model was introduced by Baker [21] who proposed an index based on suffix trees with $O(n \log n)$-time construction. The result was later improved by Cole and Hariharan [22] to $O(n)$ construction time. Recently, Lee et al. [23] presented an online construction algorithm with the same time complexity. What Cole and Hariharan [22] proposed was actually a general scheme for construction of suffix trees for so-called quasi-suffix families with a constant-time character oracle. This result can also be applied in the order-preserving setting. However, the resulting construction algorithm runs in $O(n \log n / \log \log n)$ time at least for the representation of strings used in our paper (*codes* as defined in Section 2). Here the character oracle answers queries in $O(\log n / \log \log n)$ time.

Our results. We introduce an index for order-preserving pattern matching that given a pattern of length m over an integer alphabet Σ polynomially bounded in m, in $O(m)$ time determines whether the pattern occurs in the text. The index has linear size and can be constructed in $O(n \log \log n)$ expected time (or in $O(n \log^2 \log n / \log \log \log n)$ time using a deterministic algorithm). It is based on incomplete order-preserving suffix trees (*incomplete op-suffix-trees*, in short). We also show a number of other applications of these trees, including efficient computation of: longest common factors of a number of strings (using an op-suffix-tree of multiple strings), longest previous factors in a string and squares in this model. We also introduce (complete) order-preserving suffix trees (*op-suffix-trees*) and show how they can be constructed using their incomplete counterpart in $O(n \sqrt{\log n})$ time. We provide deterministic and randomized (Las Vegas) algorithms for the word-RAM model with $\Omega(\log n)$ word size.

Structure of the paper. In Section 2 we introduce *codes* which transform a string into a sequence of integer pairs so that order-isomorphism of strings is equivalent to equality of their codes. In Section 3 we give a formal definition of a complete and an incomplete op-suffix-tree and describe their basic properties. In Sections 4 and 5 we show an $O(n \log \log n)$ construction of an incomplete op-suffix-tree. The former section contains an algorithmic toolbox that is also used in further parts of the paper. Applications of our data structure are presented in Section 6. In Section 7 we obtain a construction of complete op-suffix-trees.

2. Order-Preserving Code

Let $w = w_1 \dots w_n$ be a string of length n over an integer alphabet Σ. We assume that Σ is polynomially bounded in terms of n, i.e. $\Sigma = \{1, \dots, n^c\}$ for an integer constant c. The length of w is denoted by $n = |w|$. By $w[i..j]$ we denote the *factor* $w_i \dots w_j$. For *prefixes* and *suffixes* of w we use a shorter notation $w[..i] = w[1..i]$ and $w[i..] = w[i..n]$, respectively.

We define $\alpha(w)$ and $\beta(w)$ as the rightmost occurrence of the predecessor of w_n and the rightmost occurrence of the successor of w_n among letters of $w[..n-1]$. In particular, if w_n occurs in $w[..n-1]$, then $\alpha(w)$ and $\beta(w)$ both point to the rightmost occurrence of w_n in $w[..n-1]$. More formally:

$$\alpha(w) \text{ is the largest } j < n \text{ such that } w_j = \max\{w_k \ : \ k < n, \ w_k \leq w_n\},$$

if there is no such j, then $\alpha(w) = 0$. Similarly, we define:

$$\beta(w) \text{ is the largest } j < n \text{ such that } w_j = \min\{w_k \ : \ k < n,\ w_k \geq w_n\},$$

and $\beta(w) = 0$ if no such j exists. See Figure 1 for an example.

Figure 1: To the left: $w = 5\,2\,7\,5\,1\,4$. Here $w_6 = 4$ does not occur in $w[..5]$. We have $\alpha(w) = 2$ since $w_2 = 2$ the largest letter in $w[..5]$ that is smaller than w_6. Similarly, $\beta(w) = 4$ since $w_4 = 5$ is the rightmost occurrence of the smallest letter in $w[..5]$ that is larger than w_6. To the right: $w = 5\,2\,7\,5\,1\,5$. Here $w_6 = 5$ occurs earlier in w. Therefore $\alpha(w) = \beta(w) = 4$ both indicate the position of the rightmost such occurrence.

Two simple properties of α and β are listed in the observation below.

Observation 1. *Let w be a string of length n. Then:*

(a) For any $k \in \{1, \ldots, n-1\}$ we have

$$w_k \leq w_n \iff \alpha(w) \neq 0 \ \wedge \ w_k \leq w_{\alpha(w)}$$

and symmetrically

$$w_k \geq w_n \iff \beta(w) \neq 0 \ \wedge \ w_k \geq w_{\beta(w)}.$$

(b) If $w_n = w_k$ for some $k < n$, then $\alpha(w) = \beta(w) = \max\{k < n : w_k = w_n\}$. Conversely, if $\alpha(w) = \beta(w) \neq 0$, then $w_n = w_{\alpha(w)} = w_{\beta(w)}$.

Two strings x and y of the same length are called *order-isomorphic*, written $x \approx y$, if the relative order of letters is the same in both strings. More formally, $x \approx y$ if

$$\forall_{1 \leq i,j \leq |x|} \ x_i \leq x_j \iff y_i \leq y_j.$$

Example 2. $5\,2\,7\,5\,1\,4\,9\,4\,5 \approx 6\,4\,7\,6\,3\,5\,8\,5\,6$, see Figure 2.

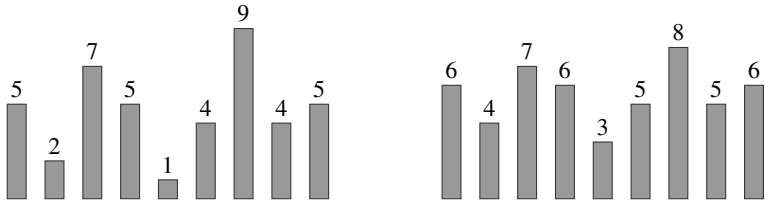

Figure 2: Example of two order-isomorphic strings. Their codes are equal to $(0,0)$ $(0,1)$ $(1,0)$ $(1,1)$ $(0,2)$ $(2,4)$ $(3,0)$ $(6,6)$ $(4,4)$.

The relation between α and β and order-isomorphism is shown in the following lemma (see also [2, 13]).

Lemma 3. *Let x and y be two strings of length n such that $x[1..n-1] \approx y[1..n-1]$. Denote $i = \alpha(x)$ and $j = \beta(x)$. If $i \neq j$, then*

$$x \approx y \iff y_i < y_n < y_j.$$

Otherwise,

$$x \approx y \iff y_i = y_n = y_j.$$

We omit conditions involving y_i or y_j when $i = 0$ or $j = 0$, respectively.

Proof. ($\Rightarrow$) Suppose that $x \approx y$. By definition, for any k, $1 \le k \le n$, the order between y_k and y_n is the same as between x_k and x_n. Thus, it suffices to prove $x_i < x_n < x_j$ and $x_i = x_n = x_j$ in the respective cases.

Definitions of $\alpha(x)$ and $\beta(x)$ yield $x_i \le x_n \le x_j$. Moreover, Observation 1(b) gives $x_i = x_n = x_j$ if $i = j$, and $x_i \ne x_n \ne x_j$ otherwise. In the latter case we conclude that $x_i < x_n < x_j$.

($\Leftarrow$) We shall prove that for any indices k, ℓ ($1 \le k < \ell \le n$) the relative order between x_k and x_ℓ is the same as between y_k and y_ℓ. If $\ell \ne n$ this follows from $x[..n-1] \approx y[..n-1]$. Hence, we may assume $\ell = n$. We consider two cases.

If $i = j$, we have $x_n = x_i$ from Observation 1(b). Combined with our assumptions—$y_i = y_n$ and $x[..n-1] \approx y[..n-1]$—this allows to conclude the claimed equivalence. Namely, the relative order between x_k and x_n is the same as between x_k and x_i, which in turn is the same as between y_k and y_i (since $i, k < n$) and consequently between y_k and y_n.

Now, we may assume $i \ne j$, which by Observation 1(b) implies $x_k \ne x_n$. If $x_k > x_n$, then $j \ne 0$ and $x_k \ge x_j$ by Observation 1(a). Thus, $y_k \ge y_j$, and consequently $y_k \ge y_j > y_n$. Analogously, if $x_k < x_n$, then $i \ne 0$, $x_k \le x_i$, and therefore $y_k \le y_i < y_n$. □

We introduce codes of strings in a similar way as in [2]:

$$\mathit{LastCode}(w) = (\alpha(w), \beta(w))$$

and

$$\mathit{Code}(w) = (\mathit{LastCode}(w[..1]),\ \mathit{LastCode}(w[..2]),\ \ldots,\ \mathit{LastCode}(w[..|w|])).$$

Using codes one can obtain an equivalent characterization of order-isomorphism:

Lemma 4. *Let x and y be two strings of length n. Then*

(a) $x \approx y \iff x[..n-1] \approx y[..n-1] \ \wedge\ \mathit{LastCode}(x) = \mathit{LastCode}(y)$.

(b) $x \approx y \iff \mathit{Code}(x) = \mathit{Code}(y)$.

Proof. (a) To prove ($\Rightarrow$) it is enough to observe that α and β depend only on the relative order of letters in the underlying string. For ($\Leftarrow$), it suffices to show that

$$x_k \le x_n \iff y_k \le y_n \quad \text{and} \quad x_k \ge x_n \iff y_k \ge y_n.$$

As x and y are symmetric, it suffices to argue that $x_k \le x_n \implies y_k \le y_n$ and $x_k \ge x_n \implies y_k \ge y_n$. If $x_k \le x_n$, then by Observation 1(a) $x_k \le x_{\alpha(x)}$. Due to $x[..n-1] \approx y[..n-1]$, we have $y_k \le y_{\alpha(x)} = y_{\alpha(y)}$. We conclude that indeed $y_k \le y_n$, again by Observation 1(a). The other implication is obtained through a symmetric argument using β instead of α.

Part (b) follows from part (a) by induction. □

The codes of strings can be computed efficiently. Applying Lemma 1 from [2] to strings over polynomially-bounded alphabet we obtain:

Lemma 5. *For a string w of length n, $\mathit{Code}(w)$ can be computed in $O(n)$ time.*

3. Order-Preserving Suffix Trees

Let us define the following family of sequences:

$$\mathit{SufCodes}(w) = \{\mathit{Code}(w[1..])\#,\ \mathit{Code}(w[2..])\#,\ \ldots,\ \mathit{Code}(w[|w|..])\#\};$$

see Figure 3. The (complete) *order-preserving suffix tree* of w (*op-suffix-tree* in short), denoted $\mathit{opSufTree}(w)$, is a compacted trie of all the sequences in $\mathit{SufCodes}(w)$.

The nodes of $\mathit{opSufTree}(w)$ with at least two children are called branching nodes. Together with the leaves they form *explicit* nodes of the tree. All the remaining nodes (dissolved in the compacted trie) are called *implicit*. By

suffixes of w:								*SufCodes*(w):								
1	2	4	4	2	5	5	1	(0,0)	(1,0)	(2,0)	(3,3)	(2,2)	(4,0)	(6,6)	(1,1)	#
2	4	4	2	5	5	1		(0,0)	(1,0)	(2,2)	(1,1)	(3,0)	(5,5)	(0,4)	#	
4	4	2	5	5	1			(0,0)	(1,1)	(0,2)	(2,0)	(4,4)	(0,3)	#		
4	2	5	5	1				(0,0)	(0,1)	(1,0)	(3,3)	(0,2)	#			
2	5	5	1					(0,0)	(1,0)	(2,2)	(0,1)	#				
5	5	1						(0,0)	(1,1)	(0,2)	#					
5	1							(0,0)	(0,1)	#						
1								(0,0)	#							

Figure 3: *SufCodes*(w) for w = 1 2 4 4 2 5 5 1.

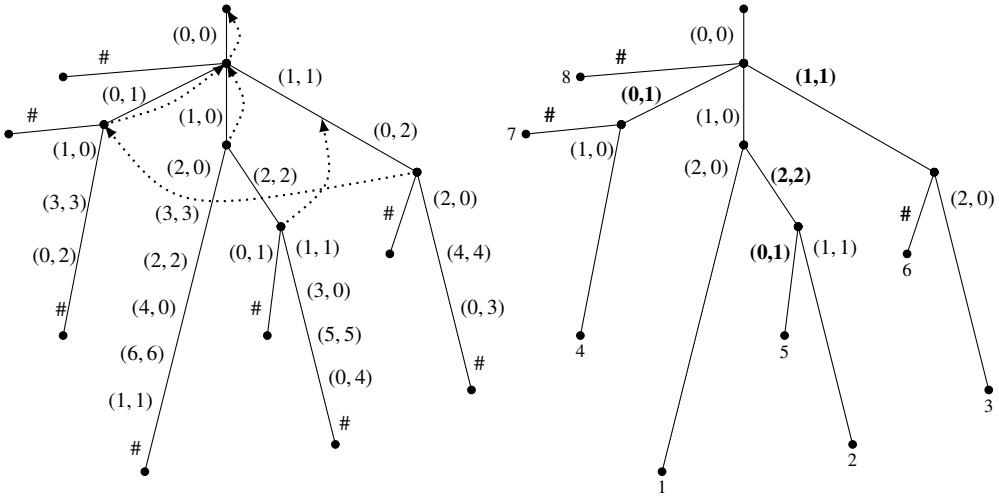

Figure 4: The uncompacted trie of *SufCodes*(w) for w = 1 2 4 4 2 5 5 1 (to the left) and its compacted version, the complete op-suffix-tree of w (to the right). The dotted arrows (left figure) show suffix links for branching nodes. One of them leads to an implicit node: *Locus*(2 5 5) = *Locus*(2 4 4) $\rightsquigarrow$ *Locus*(2 5) = *Locus*(2 4). This is because 2 5 5 1 $\not\approx$ 2 4 4 2 but 2 5 5 $\approx$ 2 4 4 and 5 5 1 $\approx$ 4 4 2. Boldface labels in the right figure are present also in the incomplete op-suffix-tree.

nodes of *opSufTree*(w) we mean both explicit and implicit nodes. For a node v, its explicit descendant (denoted as *FirstDown*(v)) is the top-most explicit node in the subtree of v (if v is explicit, then *FirstDown*(v) = v). The *locus* of v is defined as *FirstDown*(v) together with the distance between v and *FirstDown*(v). The locus of a node corresponding to x is denoted as *Locus*(x). Note that two factors share the locus precisely whenever they are order-isomorphic.

Only the explicit nodes of *opSufTree*(w) are stored. The tree contains $O(n)$ leaves. Hence, its size is $O(n)$. The leaf corresponding to *Code*($w[i..]$)# is labeled with the number i. Each branching node stores its depth and one of the occurrences of the corresponding factor. Each edge stores the code of only its first character. The codes of all the remaining characters of any edge can be obtained using a *character oracle* that can efficiently provide the code *LastCode*($w[i..j]$) for any $i < j$.

Each explicit node v stores a suffix link, *SufLink*(v), that may lead to an implicit or an explicit node (see an example in Figure 4). The suffix link is defined as:

$$\mathit{SufLink}(\mathit{Locus}(x)) = \mathit{Locus}(\mathit{DelFirst}(x)),$$

where *DelFirst*(x) results in removing the first character of x. Note that, contrary to its name, the suffix link does not literally point to the *suffix* of a node's label; see Figure 5. However, the definition is valid due to the following easy observation.

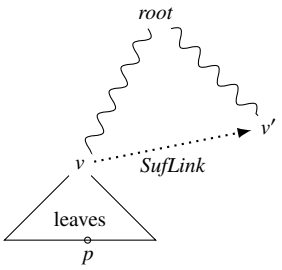

Figure 5: Let γ be the text spelled out on a path from the root to v in the uncompacted op-suffix trie of w. Similarly, let γ' be the text on a path to $v' = \mathit{SufLink}(v)$. Observe that not necessarily γ' is a suffix of γ, but $\gamma' = \mathit{Code}(\mathit{DelFirst}(x))$, where $x = w[p..p + k - 1]$ and p is the label on any leaf in the subtree rooted in v and $k = |\gamma|$ (by Observation 6, γ' is independent of the choice of a particular leaf).

Observation 6.
$\mathit{Code}(x) = \mathit{Code}(y) \implies \mathit{Code}(\mathit{DelFirst}(x)) = \mathit{Code}(\mathit{DelFirst}(y))$.

We also introduce an *incomplete* order-preserving suffix tree of w, denoted $T(w)$, in which the character oracle is not available and each explicit node v can have one outgoing edge that does not store its first character (*incomplete edge*). This edge is located on the longest path leading from v to a leaf.

4. Algorithmic Toolbox

We use a predecessor data structure to compute the *LastCodes* of a sequence changing in a queue-like manner. Dynamic predecessor queries are answered using randomization with y-fast trees introduced by Willard [24] or deterministically with exponential search trees of Andersson and Thorup [25]:

Lemma 7 ([24, 25])**.** *Let N be an integer such that $\omega = \Omega(\log N)$, where ω is the machine word-size. There exists a data structure that uses $O(|X|)$ space to maintain a set X of key-value pairs with keys from $\{1, \ldots, N\}$ and supports the following operations in $O(\log\log N)$ expected time:*

- **find**(k)*: find the value associated with k, if any,*
- **predecessor**(k)*: return the pair $(k', v) \in X$ with the largest $k' \leq k$,*
- **successor**(x)*: return the pair $(k', v) \in X$ with the smallest $k' \geq k$,*
- **remove**(k)*: remove the pair with key k,*
- **insert**(k, v)*: insert (k, v) to X removing the pair with key k, if any.*

There is also a deterministic data structure of size $O(|X|)$ which supports these queries in $O(\log^2 \log N / \log\log\log N)$ worst-case time.

Lemma 8 (Weak Character Oracle)**.** *An initially empty string x over an alphabet Σ can be maintained in a data structure $\mathcal{D}(x)$ of size $O(|x|)$ so that the following operations are supported in $O(\log\log|\Sigma|)$ expected time:*

- *compute $\mathit{LastCode}(xa)$ for a given letter $a \in \Sigma$;*
- *append a single letter $a \in \Sigma$ to x;*
- *remove the first letter from x ($\mathit{DelFirst}(x)$).*

If x is empty, the third operation is not allowed. The operations can also be supported in $O(\log^2 \log|\Sigma| / \log\log\log|\Sigma|)$ worst-case time.

Proof. We apply Lemma 7 as follows. The keys are the symbols present in x while the values associated with them are the locations of their last occurrences represented as time-stamps (that is, the ordinal numbers of the push operations used to append them). Then the *LastCode*() query is answered using one predecessor and one successor query. □

Our second tool is the dynamic weighted ancestor data structure proposed by Kopelowitz and Lewenstein [26] and originally motivated by problems related to ordinary suffix trees. A *weighted tree* is a rooted tree with integer weight assigned to each node, such that a monotonicity condition is satisfied: the weight of a node is strictly greater than the weight of its parent.

Lemma 9 ([26])**.** *Let N be an integer such that $\omega = \Omega(\log N)$, where ω is the machine word-size. There exists a data structure which maintains a weighted tree T with weights $\{1, \ldots, N\}$ in $O(|T|)$ space and supports the following operations in $O(\log \log N)$ expected time or in $O(\log^2 \log N / \log \log \log N)$ worst-case time:*

- *given a node v and a weight g find the highest ancestor of v with weight at least g,*
- *insert a leaf with weight g and v as a parent,*
- *insert a node with weight g by subdividing the edge joining v with its parent.*

The weights of inserted nodes must meet the monotonicity condition.

5. Constructing Incomplete Order-Preserving Suffix Tree

We design a version of Ukkonen's algorithm [27] in which suffix links are computed using weighted ancestor queries; see Figure 6. The weights of explicit nodes represent their depths. In this case for a node u, by $\mathit{WeightedAnc}(u, d)$ we denote its (explicit or implicit) ancestor at depth d. Note that such a node can be found with a weighted ancestor query of Lemma 9, which actually returns $\mathit{FirstDown}(\mathit{WeightedAnc}(u, d))$.

Our algorithm works online. While reading the string w it maintains:

- the incomplete op-suffix-tree $T(w)$ for w without endmarkers (#);
- the longest suffix x of w such that $\mathit{Code}(x)$ corresponds to a non-leaf node of $T(w)$, together with the data structure $\mathcal{D}(x)$; x is called the *active suffix*;
- the node (explicit or implicit) $\mathit{Locus}(x)$, called the *active node*.

In the algorithm all implicit nodes are represented in a canonical form: the explicit descendant (*FirstDown*) and the distance to this descendant (depth difference). Each explicit node stores a dynamic hash table (see [22, 28]) of its explicit children, indexed by the labels of the respective edges. The explicit child corresponding to the incomplete edge is stored outside of the hash table.

Description of one iteration of the algorithm. In one iteration w is extended by one character, say a. We traverse the so-called *active path* in $T(w)$:

1. We search for the longest suffix x' of x such that $\mathit{Locus}(x'a)$ appears in the tree.
2. For each longer suffix x'' of x we create a branch leading to a new leaf node corresponding to $x''a$.
3. The active path is found by jumping along suffix links, starting at the active node.
4. The suffix links of the newly created explicit nodes are computed using weighted ancestor queries; see Figure 6. This part differs substantially from Ukkonen's original algorithm.
5. The end-point of the active path becomes the parent of the new active node, and $x'a$ is the new active suffix.

To compute the last symbol of $\mathit{Code}(xa)$ we use the Weak Character Oracle (Lemma 8).

In the algorithm we use two auxiliary subroutines.

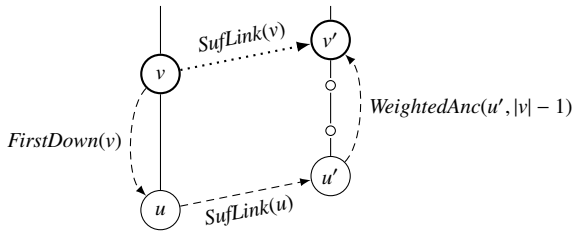

Figure 6: Computation of *SufLink*(v). Here u is explicit.

Function *Transition*$(v, (p, q))$**.** This function checks if v has an (explicit or implicit) child v' such that the edge from v to v' represents the code (p, q). It returns the node v' or **nil** if such a node does not exist. In the implementation we check, using hash tables, if any of the labeled edges outgoing from v starts with the code (p, q). For the (at most one for v) incomplete edge we can test whether the starting letter of its code equals (p, q) by verifying the inequalities from Lemma 3 for the corresponding factor of the text w.

Function *Branch*$(v, (p, q), i)$**.** This function creates an edge from v with the code (p, q). The newly created leaf is labeled with i. If v was implicit, then it is made explicit at this point. The edge leading to its already existing child remains incomplete. This procedure also adds a *SufLink* from the leaf number $i - 1$ to the new leaf number i.

The whole structure of the algorithm is presented in the following Algorithm 1.

Algorithm 1: Construct incomplete op-suffix-tree $T(w)$

```
Initialize T as a tree consisting of a single node: root;
v := root; x := empty string;
for i := 1 to n + 1 do
    if i ≤ n then lc := LastCode(x · w_i) else lc := #;
    beyondRoot := false;
    while Transition(v, lc) = nil do
        u := FirstDown(v); { the first explicit node below v, including v }
        Branch(v, lc, i − |x|);
        if v ≠ root then
            x := DelFirst(x);
            if i ≤ n then lc := LastCode(x · w_i);
            { Computation of suffix link; see Figure 6: }
            u' := SufLink(u); { u' can be an implicit node }
            v' := WeightedAnc(u', |x| − 1); { weighted ancestor query }
            SufLink(v) := v'; v := v';
        else
            beyondRoot := true;
        end
    end
    if not beyondRoot then
        v := Transition(v, lc);
        if i ≤ n then x := x · w_i;
    end
end
return T;
```

Remark 10 (Why incomplete?). At first glance it might be unclear why incomplete edges appear. Consider the situation when we jump to an implicit node $v' = \mathit{SufLink}(v)$ and we later branch in this node. The node v' becomes explicit and the existing edge from this node to some node z becomes an *incomplete edge*. Despite incompleteness of the edge (v', z) the forthcoming equality tests between the (known) last code letter of the active string and the first (unknown) code letter of the label of this edge can be done quickly due to Lemma 3.

In the pseudocode above we perform $O(n)$ operations in total. This follows from the fact that each step of the while-loop creates a new edge in the tree. The operations involving x and the operations on the data structure for weighted ancestor queries are performed in $O(\log\log n)$ expected time or in $O(\log^2 \log n / \log\log\log n)$ worst-case time each. All the remaining operations require constant time only. Hence, we obtain the following result.

Theorem 11. *The incomplete op-suffix-tree $T(w)$ for a string w of length n can be computed in $O(n \log\log n)$ expected time or in $O(n \log^2 \log n / \log\log\log n)$ worst-case time.*

6. Applications of Order-Preserving Incomplete Suffix Trees

6.1. Indexing Problem

The most common application of suffix trees is pattern matching with time complexity independent of the text length.

Theorem 12. *Assume that we have $T(w)$ for a string w of length n. Given a pattern x of length m, one can check if w contains a factor order-isomorphic to x in $O(m)$ time and report all occurrences of such factors in $O(m + Occ)$ time, where Occ is the number of occurrences reported.*

Proof. First, we compute the code of the pattern. This takes $O(m)$ time due to Lemma 5. To answer a query, we traverse down $T(w)$ using the successive symbols of the code. At each step we use the function $\mathit{Transition}(v, (p, q))$.

This enables to find the locus of x in $O(m)$ time. Afterwards all the occurrences of factors that are order-isomorphic to x can be listed in the usual way by inspecting all leaves in the subtree of $\mathit{Locus}(x)$. □

Remark 13. The $O(m)$ query time requires the letters in the pattern x to be sortable in $O(m)$ time. In general, sorting can be performed in $O(m\sqrt{\log\log m})$ expected time [29] or in $O(m \log\log m)$ time deterministically [30] since we assume that Σ consists of integers fitting into machine words.

6.2. Order-Preserving Suffix Tree of Multiple Strings

In many applications instead of a suffix tree of a single text one uses a joint suffix tree of several strings. In the standard setting such a *generalized suffix tree* of $(w^{(i)})_{i=1}^{k}$ is often defined as the suffix tree of $w^{(1)}\$_1 w^{(2)}\$_2 \dots w^{(k)}$, where $\$_i$ are distinct endmarkers. In the order-preserving setting, however, such a black-box reduction fails since the construction algorithm would use codes of the delimiters $\$_i$ instead of the delimiters themselves. Nevertheless, we can adapt the algorithm presented in the previous section to construct $T(w^{(1)}, \dots, w^{(k)})$, the incomplete generalized op-suffix-tree of $(w^{(i)})_{i=1}^{k}$.

Before we discuss the necessary adjustments, let us formally define the tree $T(w^{(1)}, \dots, w^{(k)})$. Let $\#_1, \dots, \#_k$ be distinct symbols which do not occur in $\mathit{Code}(x)$ for any string x. We define

$$\mathit{SufCodes}(w^{(1)}, \dots, w^{(k)}) = \{\mathit{Code}(w^{(i)}[j..])\#_i : 1 \le i \le k,\ 1 \le j \le |w^{(i)}|\}.$$

The generalized order-preserving suffix tree $\mathit{opSufTree}(w^{(1)}, \dots, w^{(k)})$ is a compacted trie containing all suffixes in $\mathit{SufCodes}(w^{(1)}, \dots, w^{(k)})$. The auxiliary data stored in explicit nodes is the same as in the order-preserving suffix tree of a single string. The leaf corresponding to $\mathit{Code}(w^{(i)}[j..])\#_i$ is labeled with a pair (i, j). The value i is sometimes referred to as the *color* of the leaf.

In the generalized incomplete order-preserving suffix tree $T(w^{(1)}, \dots, w^{(k)})$ each explicit node v may have one outgoing edge that does not store its first character. This is the edge leading towards the leaf with lexicographically smallest label (i, j) among all the leaves in the subtree of v.

Theorem 14. *The incomplete generalized op-suffix-tree $T(w^{(1)}, \ldots, w^{(k)})$ of a collection of strings $(w^{(i)})_{i=1}^{k}$ of total length n can be constructed in $O(n \log \log n)$ expected time or in $O(n \log^2 \log n / \log \log \log n)$ worst-case time.*

Proof. It suffices to run Algorithm 1 sequentially for all strings $w^{(i)}$ with two minor modifications:

- in the first step T is initialized as the tree consisting of the root only but later the result of the previous steps is used,
- a different end-marker $\#_i$ is appended for each string.

The number of iterations of the while-loop when processing $w^{(i)}$ is bounded by $|w^{(i)}|$, and each iteration takes $O(\log \log n)$ expected time. Thus, in total the running time is $O(n \log \log n)$ as announced. □

One of the motivating applications of suffix trees in the standard setting was finding the longest common factor of two strings. An analogue of this problem in the order-preserving setting is especially important since it provides a way to find common trends in time series. In a generalization of this problem, given an integer d and k strings $w^{(1)}, \ldots, w^{(k)}$, we need to find a longest string that is order-isomorphic to a factor of at least d out of k strings $w^{(1)}, \ldots, w^{(k)}$. An efficient solution to this problem can be obtained using the generalized op-suffix-tree.

Theorem 15. *The longest order-preserving factors common to at least d out of k given strings $w^{(1)}, \ldots, w^{(k)}$ of total length n can be computed in $O(n)$ time for all values $d = 2, \ldots, k$, provided that $T(w^{(1)}, \ldots, w^{(k)})$ is given.*

Proof. A factor common to d strings corresponds to a node in the generalized op-suffix-tree with leaves of at least d distinct colors in the subtree. Given $T(w^{(1)}, \ldots, w^{(k)})$, these numbers can be computed for all explicit nodes in $O(n)$ time using a result of Hui [31]. □

In the most natural case of $k = d = 2$, with strings of lengths $n_1 \leq n_2$, one can actually obtain an algorithm using $O(n_2 \log \log n_1)$ time and $O(n_1)$ space. This is achieved using a standard technique of partitioning the longer string into fragments of length up to $2n_1$ with overlaps of n_1 characters. The longest common order-preserving factor must occur in one of the fragments.

6.3. Longest Previous Order-Preserving Factors

Given a string w of length n, we introduce the longest previous order-preserving factor (*op*-LPF) table defined as follows. For any position i in w, *op*-LPF$[i]$ specifies the length of the longest factor u of w starting at position i such that a factor order-isomorphic to u occurs earlier in w. Formally,

$$op\text{-LPF}[i] = \max\{\ell \ : \ w[i..i+\ell-1] \approx w[j..j+\ell-1] \text{ for some } j < i\};$$

see Figure 7. Several algorithms solving this problem in linear time in the standard setting are known [32, 33].

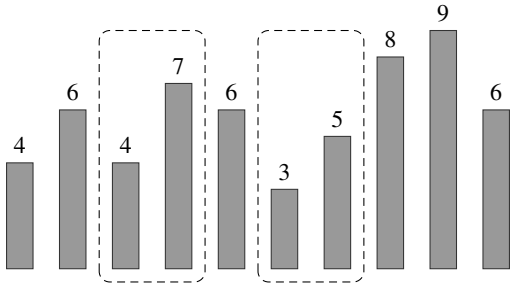

Figure 7: The *op*-LPF table for this string is: $[0, 1, 2, 2, 2, 2, 3, 2, 2, 1]$. We have *op*-LPF$[6] = 2$ because $3\,5 \approx 4\,7$; see the rectangles in the figure.

The *op*-LPF table can be computed during the construction of $T(w)$ with Algorithm 1. Every time we perform a $Branch(v, (p, q), j)$ operation to insert a new leaf, then $LPF[j]$ becomes the depth of the node v (which is equal to $|x|$ in the algorithm). Here we use the fact that the leaves are added to the tree in the order of increasing labels. Therefore,

when adding the leaf number j, all the suffixes longer than the j-th suffix are already present in the tree and we can compute the answer for j. Actually, one could apply the same approach to compute the standard *LPF* table using Ukkonen's algorithm.

While this approach has several advantages, it is likely that Algorithm 1 is not optimal and faster solutions do not need to be based on the framework due to Ukkonen. Thus, below we provide a black-box solution using the (incomplete) op-suffix tree as the starting point.

The problem of computing *LPF* table can actually be defined for arbitrary rooted trees with a linear order of the leaves: for each leaf L we are to find a leaf $L' < L$ such that $LCA(L, L')$, i.e. the lowest common ancestor of the two, is as high as possible. The actual value $LPF[L]$ is then the (weighted) depth of $LCA(L, L')$.

The classic linear-time solution is as follows. We arrange leaves in the order of the depth-first traversal of the tree: $L_1, \ldots, L_n$. Then, we observe that the nodes $LCA(L_1, L_k), \ldots, LCA(L_{k-1}, L_k)$ have non-decreasing depths while the nodes $LCA(L_{k+1}, L_k), \ldots, LCA(L_n, L_k)$ have non-increasing depths. To exploit this property, for each k we compute $p_k = \max\{k' < k : L_{k'} < L_k\}$ and $s_k = \min\{k' > k : L_{k'} < L_k\}$. Then $LPF[L_k]$ is the depth of either $LCA(L_k, L_{p_k})$ or $LCA(L_k, L_{s_k})$, whichever is larger. The sequences p_k and s_k are computed in linear time using a folklore stack-based algorithm. Once p_k and s_k are known, two *LCA* queries suffice to determine the answer $LPF[L_k]$. To conclude we need to recall that *LCA* queries in a tree can be answered in $O(1)$ time after $O(n)$-time preprocessing [34, 35].

Theorem 16. *Let w be a string of length n. Having $T(w)$, one can compute the op-LPF table of w in $O(n)$ time.*

6.4. Order-Preserving Squares

A string uv is called an *order-preserving square* (an *op-square*, in short) if $u \approx v$. The length of the op-square is $|uv|$. An op-square represents a repetition of a pattern in a time series. Using (incomplete) op-suffix-trees we can obtain efficient algorithms for finding and reporting op-squares. We show how to modify an $O(n \log n)$-time square-detecting algorithm by Gusfield and Stoye [36] to check, for each length k, if a given string w contains an op-square of length $2k$.

Note that in the standard setting the analogous problem can be solved in $O(n)$ time [37]. This approach, however, is based on an older $O(n \log n)$-time algorithm of Main and Lorentz [38]. The key property exploited by this solution is that if $w[i..i + 2k - 1]$ and $w[j..j + 2k - 1]$ are (regular) squares such that $i \leq j \leq i + k$, then $w[p..p + 2k - 1]$ is a square for every $p \in \{i, \ldots, j\}$. In the order-preserving setting this is no longer true.

Example 17. For $w = 1\,2\,5\,6\,3\,4$, the factors $w[1..4]$ and $w[3..6]$ are op-squares, but $w[2..5]$ is not because $2\,5 \not\approx 6\,3$.

Branching regular squares. We say that a substring $w[i..i + 2k - 1]$ is a *branching square* if $w[i..i + k - 1] = w[i + k..i + 2k - 1]$ and $w[i + 2k] \neq w[i]$. The algorithm of Gusfield and Stoye [36] uses the suffix tree of a text w, $|w| = n$, to find all *branching squares* in w in $O(n \log n)$ time. A branching square of length $2k$ (i.e., $w[i..i + 2k - 1]$) is detected as a pair of leaves with labels that differ by k (i.e., i and $i + k$) whose lowest common ancestor is at depth k (and corresponds to $w[i..i + k - 1]$).

Non-extendible and non-shiftable op-squares. We say that an op-square $w[i..i + 2k - 1]$ is *non-extendible* if either $i + 2k - 1 = n$ or

$$w[i..i + k] \not\approx w[i + k..i + 2k].$$

A *non-shiftable* op-square is defined similarly but with the condition:

$$w[i + 1..i + k] \not\approx w[i + k + 1..i + 2k].$$

Both notions are generalizations of branching regular squares to the order-preserving setting. It turns out that when we apply the algorithm from [36] to the op-suffix-tree of w, we find all non-extendible op-squares in w (i.e., not necessarily all non-shiftable op-squares); see the following lemma.

Lemma 18. *Let w be a string. Then $w[i..i + 2k - 1]$ is a non-extendible op-square if and only if the LCA of leaves labeled with i and $i + k$ in $T(w)$ is Locus$(w[i..i + k - 1])$.*

Proof. ($\Rightarrow$) If $w[i..i+2k-1]$ is a non-extendible op-square, then the longest common prefix of $Code(w[i..i+k])$ and $Code(w[i+k..i+2k])$ is exactly $Code(w[i..i+k-1])$. This yields that indeed $Locus(w[i..i+k-1])$ is the lowest common ancestor of the leaves labeled i and $i+k$.

($\Leftarrow$) If the leaves labeled with i and $i+k$ have their lowest common ancestor at a node at depth exactly k, then

$$Code(w[i..i+k-1]) = Code(w[i+k..i+2k-1])$$

but

$$Code(w[i..i+k]) \neq Code(w[i+k..i+2k]).$$

Hence, indeed $w[i..i+2k-1]$ is a non-extendible op-square. □

Now, it suffices to prove the following property.

Lemma 19. *If w contains an op-square of a given length, then it contains a non-extendible op-square of the same length.*

Proof. Let $w[i..i+2k-1]$ be the rightmost op-square of length $2k$ in w. If $i+2k-1 = n$, then it is already a non-extendible op-square. Otherwise, it is a non-shiftable op-square:

$$w[i+1..i+k] \not\approx w[i+k+1..i+2k].$$

Hence,

$$w[i..i+k] \not\approx w[i+k..i+2k]$$

and consequently $w[i..i+2k-1]$ is a non-extendible op-square. □

Consequently, we obtain an efficient algorithm for detecting an op-square of a given length. Note that the algorithm does not require to query the character oracle. It only processes the skeleton of the suffix tree.

Theorem 20. *For a string w of length n, after $O(n \log n)$-time preprocessing one can check if w contains an op-square of a given length in $O(1)$ time.*

The algorithm of Gusfield and Stoye can also compute all the occurrences of regular squares in a string in additional time proportional to the number of reported occurrences. For this, it starts at every branching square $w[i..i+2k-1]$ and shifts it to the left position-by-position as long as it forms a square, i.e. as long as $w[i-j] = w[i+k-j]$, $j = 1, 2, \ldots$

A generalization of this algorithm to op-squares requires efficient testing if an op-square can be shifted to the left. This could be done using the character oracle for the reversed text. However, there is a more efficient solution.

Theorem 21. *All occurrences of order-preserving squares in a string w of length n can be computed in $O(n \log n + Occ)$ time, where Occ is the total number of occurrences of op-squares.*

Proof. We use the fact that the string $w[i..i+2k-1]$ is an op-square if and only if the LCA node of the leaves of $T(w)$ with labels i and $i+k$ has depth at least k.

Recall that after $O(n)$ preprocessing, LCA of nodes in a tree can be computed in $O(1)$ time [34, 35]. Using LCA-queries we can keep shifting an non-extendible op-square to the left. We stop either when the tested substring is not an op-square or when we encounter another non-extendible op-square. The latter situation is possible since non-extendible op-squares can still be shiftable. We obtain an algorithm with required complexity. □

7. Constructing Complete Order-Preserving Suffix Tree

In this section we present efficient construction algorithms for a complete op-suffix-tree in two variants. In the first variant we use the codes from Section 2 and obtain $O(\frac{n \log n}{\log \log n})$ construction time. Later, we choose another code to express order-isomorphism to obtain $O(n \sqrt{\log n})$-time construction of an op-suffix-tree that uses this code.

In the first variant we apply the following result by Babenko et al. [39], a data structure for *range rank* and *range selection* queries.

Lemma 22 ([39])**.** *An array $A[1..n]$ of integers (fitting machine words) can be preprocessed in $O(n\sqrt{\log n})$ time so that one can answer the following queries in $O(\frac{\log n}{\log\log n})$ time:*

1. *Given indices i, j, k (with $i \le j$) count the number of elements in $A[i..j]$ smaller than $A[k]$.*
2. *Given indices $i \le j$ and an integer k, $1 \le k \le j - i + 1$, find the index of the k-th smallest element in $A[i..j]$.*

As a consequence, we obtain a data structure for *range predecessor* and *range successor* queries.

Corollary 23. *An array $A[1..n]$ of integers in $\{1, \ldots, n^{O(1)}\}$ can be preprocessed in $O(n\sqrt{\log n})$ time so that one can answer the following queries in $O(\frac{\log n}{\log\log n})$ time:*

1. *Given indices $i \le j$ and an integer v compute the index of the largest element in $A[i..j]$ not larger than v.*
2. *Given indices $i \le j$ and an integer v compute the index of the smallest element in $A[i..j]$ not smaller than v.*

Proof. We maintain the data structure of Lemma 22 as well as the data structure for (static) predecessor and successor queries, which maps any value present in A into an index where it occurs. For this, we may use exponential search trees; see Lemma 7. This gives additional $o(n\log^2\log n)$ in construction time and $o(\log^2\log n)$ for each query.

In order to answer a predecessor query for v in range $[i..j]$ we proceed as follows: we find an index k such that $A[k]$ is the successor of $v + 1$ in $A[1..n]$ and ask for a rank of $A[k]$ in $A[i..j]$. If the result r is non-zero, we return the r-th smallest element in $A[i..j]$. Otherwise, v has no predecessor in $A[i..j]$. The procedure for a successor query is analogous. □

Note that data structures with faster query times for range predecessor/successor problem are known [40, 41]. However, the construction times of these data structures are $\Omega(n\log n)$.

The character oracle of Lemma 8 is efficient but it allows computation of *LastCodes* only for a dynamic string that changes in a queue-like manner. Now we show a general character oracle that is able to compute the *LastCode* for any factor of w.

Lemma 24 (Strong Character Oracle)**.** *A string w of length n can be preprocessed in $O(n\sqrt{\log n})$ time, so that given indices $i \le j$ one can compute $\mathit{LastCode}(w[i..j])$ in $O(\frac{\log n}{\log\log n})$ time.*

Proof. We construct an array $A[1..n]$ with $A[k] = (w_k, k)$ and build the structure of Corollary 23 over this array. (Actually, we map $(a, b) \mapsto an + b$ so that the values are integers.) In order to answer the $\alpha(w[i..j])$ query it suffices to compute the index k_1 of the predecessor of (w_j, n) in $A[i..j-1]$. We have $\alpha(w[i..j]) = k_1 - i + 1$. The $\beta(w[i..j])$ values are computed similarly using range successor queries in an array $B[1..n]$ with $B[k] = (w_k, -k)$. □

To obtain a complete op-suffix-tree, we need to put labels on incomplete edges and to provide a character oracle. Note that, using a character oracle working in $f(n)$ time, we can fill in the missing labels in $O(nf(n))$ time.

Corollary 25. *The op-suffix-tree of a string of length n can be constructed in $O(\frac{n\log n}{\log\log n})$ time.*

7.1. Faster Construction with Different Codes

Below we show a slightly faster construction. For this, however, we need a different encoding of strings that also maps order-isomorphism into equality. A very similar code was already presented in [1]. For a string w of length n we define:

$$\mathit{prev}_<(w) = |\{k \,:\, k < i,\ w_k < w_n\}|, \quad \mathit{prev}_>(w) = |\{k \,:\, k < i,\ w_k > w_n\}|.$$

The *counting code* of a string w is defined as

$$\mathit{LastCode}'(w) = (\mathit{prev}_<(w), \mathit{prev}_>(w))$$

and

$$\mathit{Code}'(w) = (\mathit{LastCode}'(w[..1]),\ \mathit{LastCode}'(w[..2]),\ \ldots,\ \mathit{LastCode}'(w[..|w|])).$$

Example 26. The counting code of the string 5 2 7 5 1 4 9 4 5 from Figure 2 is

$$(0,0)\,(0,1)\,(2,0)\,(1,1)\,(0,4)\,(2,3)\,(6,0)\,(2,4)\,(4,2).$$

The following lemma states that *Code′* is also an order-preserving code. A similar result is present in [1] but we provide a proof for completeness.

Lemma 27. *Let x and y be two strings of length n. Then*

(a) $x \approx y \iff x[..n-1] \approx y[..n-1] \wedge \mathit{LastCode}'(x) = \mathit{LastCode}'(y)$.

(b) $x \approx y \iff \mathit{Code}'(x) = \mathit{Code}'(y)$.

Proof. (a) To prove the ($\Rightarrow$) implication it is enough to observe that $prev_<$ and $prev_>$ depend only on the relative order of letters in the underlying string. For ($\Leftarrow$) we need to prove that the relative order between x_k and x_n is the same as between y_k and y_n for every $k < n$.

First, suppose that $x_k < x_n$ and (for a proof by contradiction) $y_k \geq y_n$. Since $prev_<(x) = prev_<(y)$, there must exist an index $\ell < n$ such that $x_\ell \geq x_n$ and $y_\ell < y_n$. This, however, implies $x_k < x_\ell$ and $y_k > y_\ell$, a contradiction with $x[..n-1] \approx y[..n-1]$. Consequently, $y_k < y_n$ whenever $x_k < x_n$. Applying a symmetric argument with $prev_>$, we conclude that $x_k > x_n$ implies $y_k > y_n$. When we exchange the roles of x and y in the previous argument, we obtain equivalences $x_k < x_n \iff y_k < y_n$ and $x_k > x_n \iff y_k > y_n$. This implies $x_k = x_n \iff y_k = y_n$ which concludes the proof that $x \approx y$.

Part (b) follows from part (a) by induction. □

The main advantage of the counting codes is the existence of an efficient *offline* character oracle, which can answer q queries about factors of a text in $O((n+q)\sqrt{\log n})$ time. To design the oracle we use a geometric approach: the computation of *LastCode′* corresponds to counting points in certain rectangles in the plane.

The orthogonal range counting problem is defined as follows. We are given n points in the plane and we are to count the number of points in axis-aligned rectangles given as queries. An efficient solution to this problem was given by Chan and Pătraşcu.

Lemma 28 (Corollary 2.3 in [42])**.** *Given n points and n axis-aligned rectangles in the plane we can count the number of points inside each rectangle in $O(n\sqrt{\log n})$ total time.*

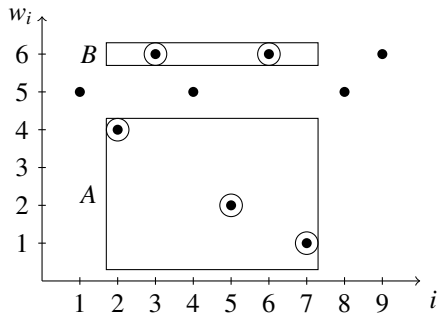

Figure 8: Geometric illustration of the sequence w = 5 4 6 5 2 6 1 5 6. The elements w_i are represented as points (i, w_i). The computation of $\mathit{LastCode}'(w[2..8]) = (3,2)$ corresponds to counting points in rectangles A, B.

Lemma 29 (Offline Character Oracle)**.** *Let w be a string length n. In $O((n+q)\sqrt{\log n})$ total time one can answer q queries asking to compute $\mathit{LastCode}'(w[i..j])$ for given indices $i \leq j$.*

Proof. Let us represent pairs (i, w_i) as points in the plane. Then we have $\mathit{LastCode}'(w[i..j]) = (a, b)$, where a is the number of points that lie within the rectangle $A = [i, j-1] \times (-\infty, w_j)$ and b is the number of points in the rectangle $B = [i, j-1] \times (w_j, \infty)$; see Figure 8.

By Lemma 28, we can count the number of points in these rectangles in $O((n+q)\sqrt{\log n})$ total time. □

Theorem 30. *The complete op-suffix-tree of a string of length n using the counting code can be constructed in* $O(n\sqrt{\log n})$ *time.*

Proof. The *skeleton* of the op-suffix-tree for each valid order-preserving code is the same. Hence, to construct the op-suffix-tree for the counting code, we compute the skeleton of the suffix tree using the deterministic algorithm for incomplete op-suffix-tree for the code of Section 2 (Theorem 11). Afterwards, we discard the original labels and use the offline character oracle to insert the first characters on each edge of the skeleton. □

8. Conclusions and Open Problems

We have presented a construction algorithm of an incomplete order-preserving suffix tree which comes in two variants: an $O(n \log\log n)$-time randomized one and its deterministic counterpart running in $O(n \log^2 \log n / \log\log\log n)$ time. We have also shown that the data structures can serve for the purposes of indexing and detecting several types of regularities in a string in the order-preserving setting.

Additionally, we have given a deterministic $O(n\sqrt{\log n})$-time construction of a complete order-preserving suffix tree. This algorithm required changing the so-called *character oracle* (that is, the encoding of the strings used to map order-isomorphism into equality). The reason was that computing the main character oracle considered in this work reduced to range predecessor/successor queries, which can be answered in $O(\log n / \log\log n)$ time, whereas computing the other oracle reduces to orthogonal range counting queries, and q such queries can be answered in $O((n+q)\sqrt{\log n})$ time. However, due to a very recent work [43], q range predecessor/successor queries can also be answered in $O((n+q)\sqrt{\log n})$ time. Consequently, one can obtain the better time complexity of constructing the complete op-suffix-tree without changing the oracle.

A number of open questions arise from our work. The most natural question refers to the existence of faster construction algorithms of both data structures or deterministic construction algorithms with the same time complexities. Another problem is related to order-preserving indexing. Our index allows for $O(m)$-time queries, where m is the length of the pattern, assuming that the alphabet of the pattern is polynomial in m. One can ask whether there exists an index with equally good construction time and $O(m)$-time queries for patterns over larger alphabet, i.e., alphabet that is polynomial in n. Finally, it would be interesting to know if there is an $o(n \log n)$-time algorithm for finding the longest order-preserving square in a string.

Acknowledgements

Tomasz Kociumaka is supported by Polish budget funds for science in 2013–2017 as a research project under the 'Diamond Grant' program (Ministry of Science and Higher Education, Republic of Poland, grant number DI2012 01794). Jakub Radoszewski receives financial support of Foundation for Polish Science and is supported by the Polish Ministry of Science and Higher Education under the 'Iuventus Plus' program in 2015-2016 grant no. 0392/IP3/2015/73. Wojciech Rytter is supported by grant no. NCN2014/13/B/ST6/00770 of the National Science Centre.

References

[1] J. Kim, P. Eades, R. Fleischer, S.-H. Hong, C. S. Iliopoulos, K. Park, S. J. Puglisi, T. Tokuyama, Order-preserving matching, Theor. Comput. Sci. 525 (2014) 68–79.

[2] M. Kubica, T. Kulczyński, J. Radoszewski, W. Rytter, T. Waleń, A linear time algorithm for consecutive permutation pattern matching, Inf. Process. Lett. 113 (12) (2013) 430–433.

[3] D. E. Knuth, The Art of Computer Programming, Volume I: Fundamental Algorithms, 3rd Edition, Addison-Wesley, 1997.

[4] L. Lovász, Combinatorial problems and exercices, North-Holland, 1979.

[5] D. Rotem, Stack sortable permutations, Discrete Math. 33 (2) (1981) 185–196.

[6] P. Bose, J. F. Buss, A. Lubiw, Pattern matching for permutations, Inf. Process. Lett. 65 (5) (1998) 277–283.
[7] M. H. Albert, R. E. L. Aldred, M. D. Atkinson, D. A. Holton, Algorithms for pattern involvement in permutations, in: P. Eades, T. Takaoka (Eds.), Algorithms and Computation — ISAAC 2001, Vol. 2223 of LNCS, Springer Berlin Heidelberg, 2001, pp. 355–366.
[8] S. Guillemot, S. Vialette, Pattern matching for 321-avoiding permutations, in: Y. Dong, D.-Z. Du, O. H. Ibarra (Eds.), Algorithms and Computation — ISAAC 2009, Vol. 5878 of LNCS, Springer Berlin Heidelberg, 2009, pp. 1064–1073.
[9] L. Ibarra, Finding pattern matchings for permutations, Inf. Process. Lett. 61 (6) (1997) 293–295.
[10] S. Guillemot, D. Marx, Finding small patterns in permutations in linear time, in: C. Chekuri (Ed.), 25th Annual ACM-SIAM Symposium on Discrete Algorithms — SODA 2014, SIAM, 2014, pp. 82–101.
[11] M. Bruner, M. Lackner, The computational landscape of permutation patterns, J. Pure Appl. Math. 24 (2) (2013) 83–101.
[12] D. Belazzougui, A. Pierrot, M. Raffinot, S. Vialette, Single and multiple consecutive permutation motif search, in: L. Cai, S. Cheng, T. W. Lam (Eds.), Algorithms and Computation — ISAAC 2013, Vol. 8283 of LNCS, Springer Berlin Heidelberg, 2013, pp. 66–77.
[13] S. Cho, J. C. Na, K. Park, J. S. Sim, A fast algorithm for order-preserving pattern matching, Inf. Process. Lett. 115 (2) (2015) 397–402.
[14] T. Chhabra, J. Tarhio, Order-preserving matching with filtration, in: J. Gudmundsson, J. Katajainen (Eds.), Experimental Algorithms — SEA 2014, Vol. 8504 of LNCS, Springer International Publishing, 2014, pp. 307–314.
[15] S. Faro, M. O. Külekci, Efficient algorithms for the order preserving pattern matching problem, CoRR abs/1501.04001.
[16] M. M. Hasan, A. S. M. S. Islam, M. S. Rahman, M. S. Rahman, Order preserving pattern matching revisited, Pattern Recogn. Lett. 55 (2015) 15–21.
[17] P. Gawrychowski, P. Uznański, Order-preserving pattern matching with k mismatches, in: A. S. Kulikov, S. O. Kuznetsov, P. A. Pevzner (Eds.), Combinatorial Pattern Matching — CPM 2014, Vol. 8486 of LNCS, Springer International Publishng, 2014, pp. 130–139.
[18] M. M. Hasan, A. S. M. S. Islam, M. S. Rahman, M. S. Rahman, Order preserving prefix tables, in: E. S. de Moura, M. Crochemore (Eds.), String Processing and Information Retrieval — SPIRE 2014, Springer International Publishing, 2014, pp. 111–116.
[19] J. Kim, A. Amir, J. C. Na, K. Park, J. S. Sim, On representations of ternary order relations in numeric strings, in: C. S. Iliopoulos, A. Langiu (Eds.), Algorithms for Big Data — IABCD 2014, Vol. 1146 of CEUR Workshop Proceedings, CEUR-WS.org, 2014, pp. 46–52.
[20] T. Kociumaka, J. Radoszewski, W. Rytter, T. Waleń, Maximum number of distinct and nonequivalent nonstandard squares in a word, in: A. M. Shur, M. V. Volkov (Eds.), Developments in Language Theory — DLT 2014, Vol. 8633 of LNCS, Springer International Publishing, 2014, pp. 215–226.
[21] B. S. Baker, Parameterized pattern matching: Algorithms and applications, J. Comput. Syst. Sci. 52 (1) (1996) 28–42.
[22] R. Cole, R. Hariharan, Faster suffix tree construction with missing suffix links, SIAM J. Comput. 33 (1) (2003) 26–42.
[23] T. Lee, J. C. Na, K. Park, On-line construction of parameterized suffix trees for large alphabets, Inf. Process. Lett. 111 (5) (2011) 201–207.
[24] D. E. Willard, Log-logarithmic worst-case range queries are possible in space $\Theta(N)$, Inf. Process. Lett. 17 (2) (1983) 81–84.
[25] A. Andersson, M. Thorup, Dynamic ordered sets with exponential search trees, J. ACM 54 (3) (2007) 13.
[26] T. Kopelowitz, M. Lewenstein, Dynamic weighted ancestors, in: N. Bansal, K. Pruhs, C. Stein (Eds.), 18th Annual ACM-SIAM Symposium on Discrete Algorithms — SODA 2007, SIAM, 2007, pp. 565–574.
[27] E. Ukkonen, On-line construction of suffix trees, Algorithmica 14 (3) (1995) 249–260.
[28] M. Dietzfelbinger, A. R. Karlin, K. Mehlhorn, F. Meyer auf der Heide, H. Rohnert, R. E. Tarjan, Dynamic perfect hashing: Upper and lower bounds, SIAM J. Comput. 23 (4) (1994) 738–761.
[29] Y. Han, M. Thorup, Integer sorting in $O(n\sqrt{\log\log n})$ expected time and linear space, in: 43rd Symposium on Foundations of Computer Science — FOCS 2002, IEEE Computer Society, 2002, pp. 135–144.
[30] Y. Han, Deterministic sorting in $O(n \log\log n)$ time and linear space, J. Algorithms 50 (1) (2004) 96–105.
[31] L. C. K. Hui, Color set size problem with application to string matching, in: A. Apostolico, M. Crochemore, Z. Galil, U. Manber (Eds.), Combinatorial Pattern Matching — CPM 1992, Vol. 644 of LNCS, Springer Berlin Heidelberg, 1992, pp. 230–243.
[32] M. Rodeh, V. R. Pratt, S. Even, Linear algorithm for data compression via string matching, J. ACM 28 (1) (1981) 16–24.
[33] M. Crochemore, L. Ilie, C. S. Iliopoulos, M. Kubica, W. Rytter, T. Waleń, Computing the longest previous factor, Eur. J. Comb. 34 (1) (2013) 15–26.
[34] D. Harel, R. E. Tarjan, Fast algorithms for finding nearest common ancestors, SIAM J. Comput. 13 (2) (1984) 338–355.
[35] M. A. Bender, M. Farach-Colton, The LCA problem revisited, in: G. H. Gonnet, D. Panario, A. Viola (Eds.), Latin American Symposium on Theoretical Informatics — LATIN 2000, Vol. 1776 of LNCS, Springer Berlin Heidelberg, 2000, pp. 88–94.
[36] J. Stoye, D. Gusfield, Simple and flexible detection of contiguous repeats using a suffix tree, Theor. Comput. Sci. 270 (1-2) (2002) 843–856.
[37] D. Gusfield, J. Stoye, Linear time algorithms for finding and representing all the tandem repeats in a string, J. Comput. Syst. Sci. 69 (4) (2004) 525–546.
[38] M. G. Main, R. J. Lorentz, An $O(n \log n)$ algorithm for finding all repetitions in a string, J. Algorithms 5 (3) (1984) 422–432.
[39] M. Babenko, P. Gawrychowski, T. Kociumaka, T. Starikovskaya, Wavelet trees meet suffix trees, in: P. Indyk (Ed.), 26th Annual ACM-SIAM Symposium on Discrete Algorithms — SODA 2015, SIAM, 2015, pp. 572–591.
[40] Y. Nekrich, G. Navarro, Sorted range reporting, in: F. V. Fomin, P. Kaski (Eds.), Algorithm Theory — SWAT 2012, Vol. 7357 of LNCS, Springer Berlin Heidelberg, 2012, pp. 271–282.
[41] G. Zhou, Sorted range reporting revisited, CoRR abs/1308.3326.
[42] T. M. Chan, M. Pătraşcu, Counting inversions, offline orthogonal range counting, and related problems, in: M. Charikar (Ed.), 21st Annual ACM-SIAM Symposium on Discrete Algorithms — SODA 2010, SIAM, 2010, pp. 161–173.
[43] M. A. Babenko, P. Gawrychowski, T. Kociumaka, T. Starikovskaya, Wavelet trees meet suffix trees, CoRR abs/1408.6182v4.